Jacobs University Bremen

CO-522-B Communications Basics Lab

Fall 2021

Lab Experiment 3: Delay and FIR

Priontu Chowdhury

Introduction

In this lab, we learn about buffer-oriented signal processing. We use MATLAB to create prototypes for a simple delay block and an RC FIR filter. We learned about two characteristics of implementation of a signal processing algorithm:

One of them is a non-real-time implementation:
1. All of the input samples are available at once.
2. The exact time for the algorithm to run is not critical.

The other is a real-time implementation:
1. A continuous stream of signals or data is received, which must also be processed continuously.
2. Timing is critical, and the algorithm must not introduce too much latency, defined loosely as the input to output delay of the system.

We then learned the use of a buffer-based strategy to develop real-time implementations of a delay block and an FIR filter block. On running the code, we were able to see how results for non-real-time implementations differ from real-time implementations. We also obtained an understanding of circular queues and its uses in these implementations acting as a buffer. We learned about latency and throughput, and how buffer size should be chosen in such a way so as to provide the most efficient implementation of the buffer.

Execution

Delay Block:

The source code for each part of the delay block is provided below:

We initialize the state using the following code:

delay_init.m

```
function [state] = delay_init(Nmax, N)
% [state] = delay_init(Nmax, N);
%
% Initializes a delay block.
%
% Inputs:
% Nmax Maximum delay supported by this block.
% N Initial delay
% Outputs:
% state State of block
% Notes:
% For this block to operate correctly,
% you should not pass in more than Nmax
% samples at a time.
%% 1. Save parameters
state.Nmax = Nmax;
% Store initial desired delay.
state.N = N;
```

```
%% 2. Create state variables
% Make the size of the buffer at least twice of the maximum delay.
% Allows us to copy in and then read out in just two steps.
state.M = 2^(ceil(log2(Nmax))+1);
% Get mask allowing us to wrap index easily

state.Mmask = state.M-1;
% Temporary storage for circular buffer
state.buff = zeros(state.M, 1);
% Set initial head and tail of buffer
state.n_h = 0;
state.n_t = N;
```

Now, we develop the delay function:

delay.m

```
function [state_out, y] = delay(state_in, x)
% [state_out, y] = delay(state_in, x);
%
% Delays a signal by the specified number of samples.
%
% Inputs:
% state_in Input state
% x Input buffer of samples

% Outputs:
% state_out Output state
% y Output buffer of samples
% Get input state
s = state_in;
% Copy in samples at tail
for ii=0:length(x)-1
 % Store a sample
 s.buff(s.n_t+1) = x(ii+1);
 % Increment head index (circular)
 s.n_t = bitand(s.n_t+1, s.Mmask);
end
% Get samples out from head
y = zeros(size(x));
for ii=0:length(y)-1
 % Get a sample
 y(ii+1) = s.buff(s.n_h+1);
 % Increment tail index
 s.n_h = bitand(s.n_h+1, s.Mmask);
end
% Output the updated state
state_out = s;
```

Testing the delay function:

Test_delay1.m

```
% test_delay1.m
%
% Script to test the delay block. Set up to model the way
% samples would be processed in a DSP program.
% Global parameters
Nb = 10; % Number of buffers
```

```
Ns = 128; % Samples in each buffer
Nmax = 200; % Maximum delay
Nd = 10; % Delay of block
% Initialize the delay block
state_delay1 = delay_init(Nmax, Nd);

% Generate some random samples.
x = randn(Ns*Nb, 1);
% Reshape into buffers
xb = reshape(x, Ns, Nb);
% Output samples
yb = zeros(Ns, Nb);
% Process each buffer
for bi=1:Nb
 [state_delay1, yb(:,bi)] = delay(state_delay1, xb(:,bi));
end
% Convert individual buffers back into a contiguous signal.
y = reshape(yb, Ns*Nb, 1);
% Check if it worked right
n = [0:(length(x)-1)];
figure(1);
plot(n, x, n, y);
figure(2);
plot(n+Nd, x, n, y, 'x');
% Do a check and give a warning if it is not right. Skip first buffer
%in check
% to avoid initial conditions.
n_chk = 1+[Ns:(Nb-1)*Ns-1];
if any(x(n_chk - Nd) ~= y(n_chk))
 warning('A mismatch was encountered.');
end
```

The results of simulation have been provided below:

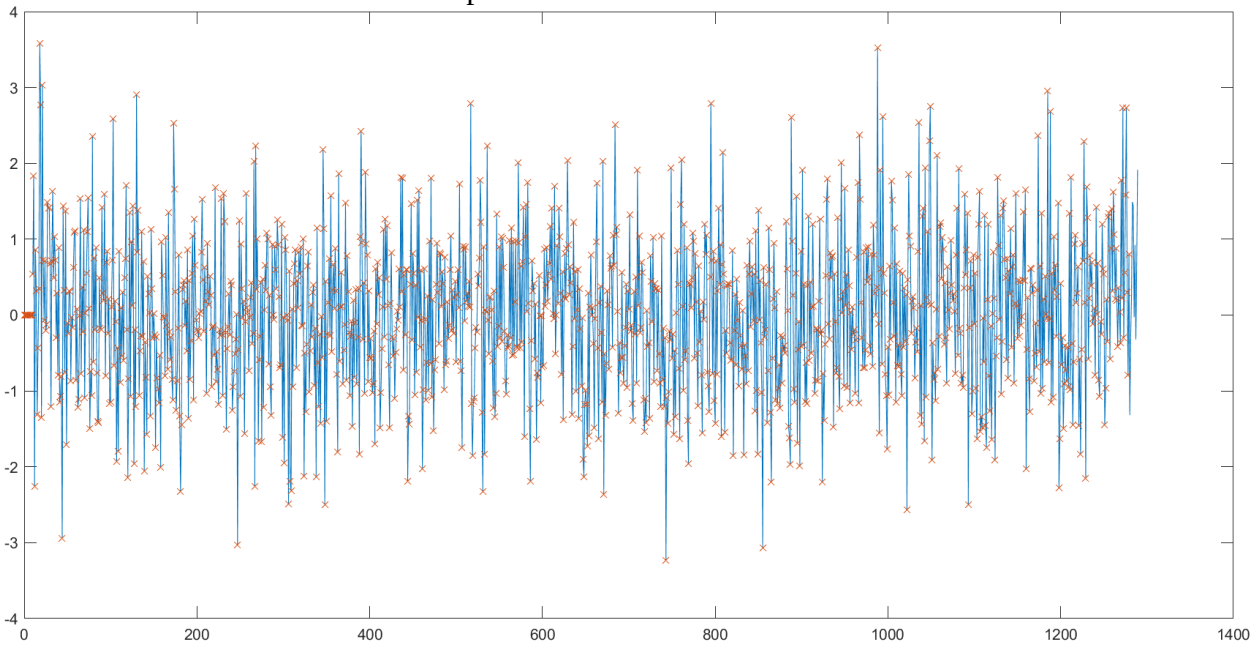

Figure: Simulation results – Plot 1

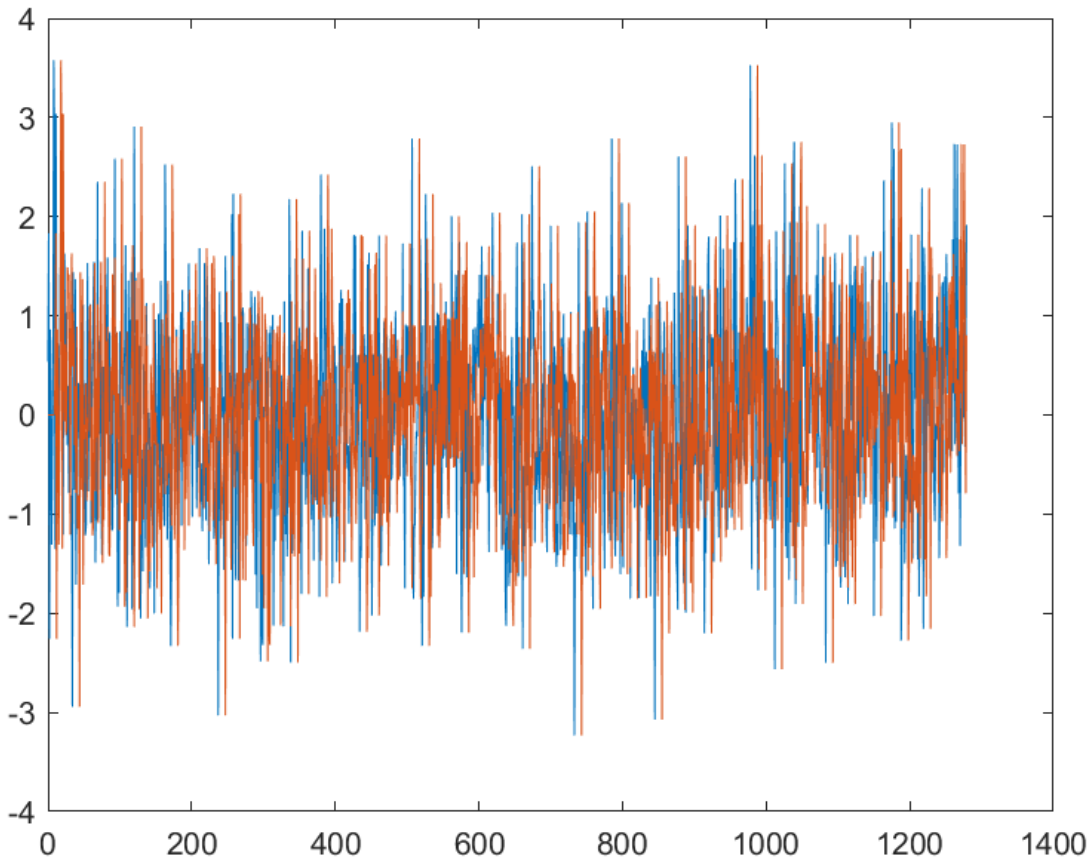

Figure: Simulation results – Plot 2

Testing with cascading delay blocks, using two different variables:
test_delay2.m

```
%
% Script to test the delay block. Set up to model the way
% samples would be processed in a DSP program.
% Global parameters
Nb = 10; % Number of buffers
Ns = 128; % Samples in each buffer
Nmax = 200; % Maximum delay
Nd = 20; % Delay of block
% Initialize the delay block
state_delay1 = delay_init(Nmax, Nd);
state_delay2 = delay_init(Nmax, Nd);
% Generate some random samples.
x = randn(Ns*Nb, 1);
% Reshape into buffers
xb = reshape(x, Ns, Nb);
% Output samples
yb = zeros(Ns, Nb);
yb2 = zeros(Ns, Nb);
% Process each buffer
for bi=1:Nb
 [state_delay1, yb(:,bi)] = delay(state_delay1, xb(:,bi));
 [state_delay2, yb2(:,bi)] = delay(state_delay2, yb(:,bi));
end
```

```
% Convert individual buffers back into a contiguous signal.
y = reshape(yb2, Ns*Nb, 1);
% Check if it worked right
n = [0:(length(x)-1)];
figure(1);
plot(n, x, n, y);
figure(2);
plot(n+Nd+Nd, x, n, y, 'x');
% Do a check and give a warning if it is not right. Skip first buffer
%in check
% to avoid initial conditions.
n_chk = 1+[Ns:(Nb-1)*Ns-1];
if any(x(n_chk - Nd) ~= y(n_chk))
 warning('A mismatch was encountered.');
end
```

The simulation results have been provided below:

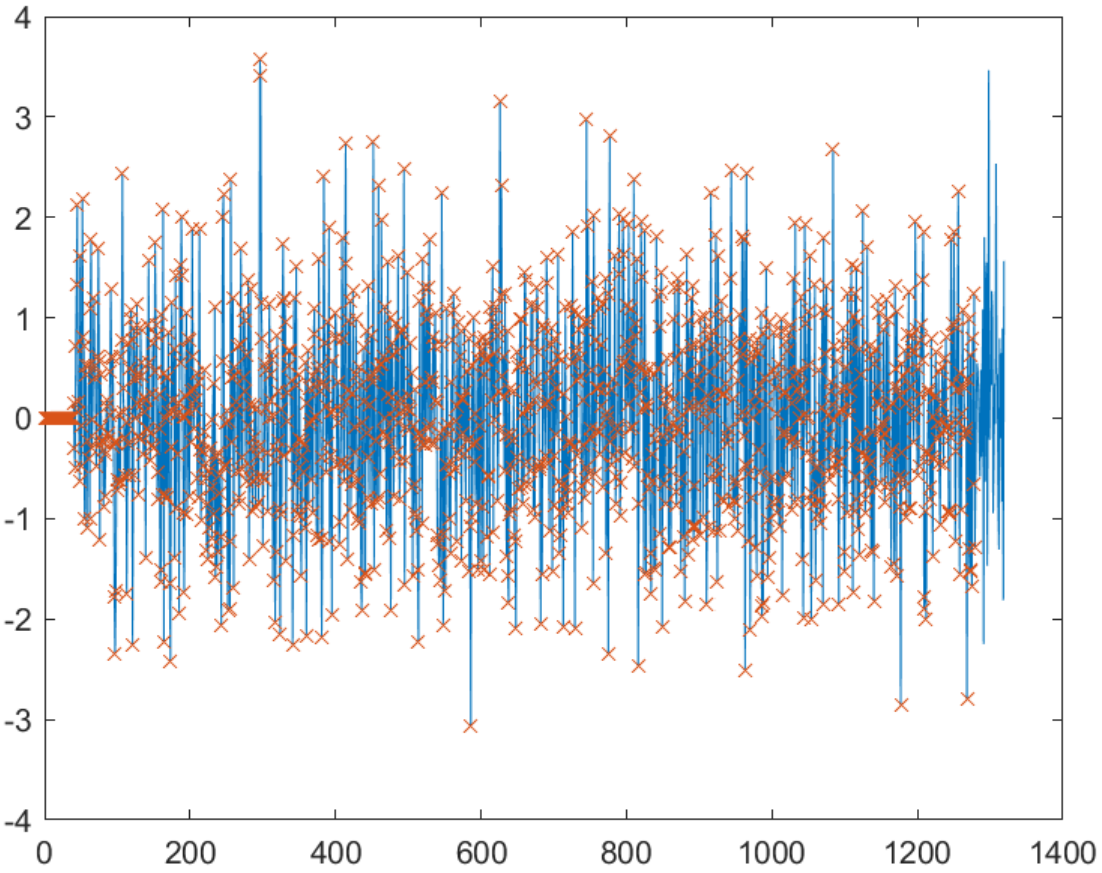

Figure: Simulation Results – Plot 1

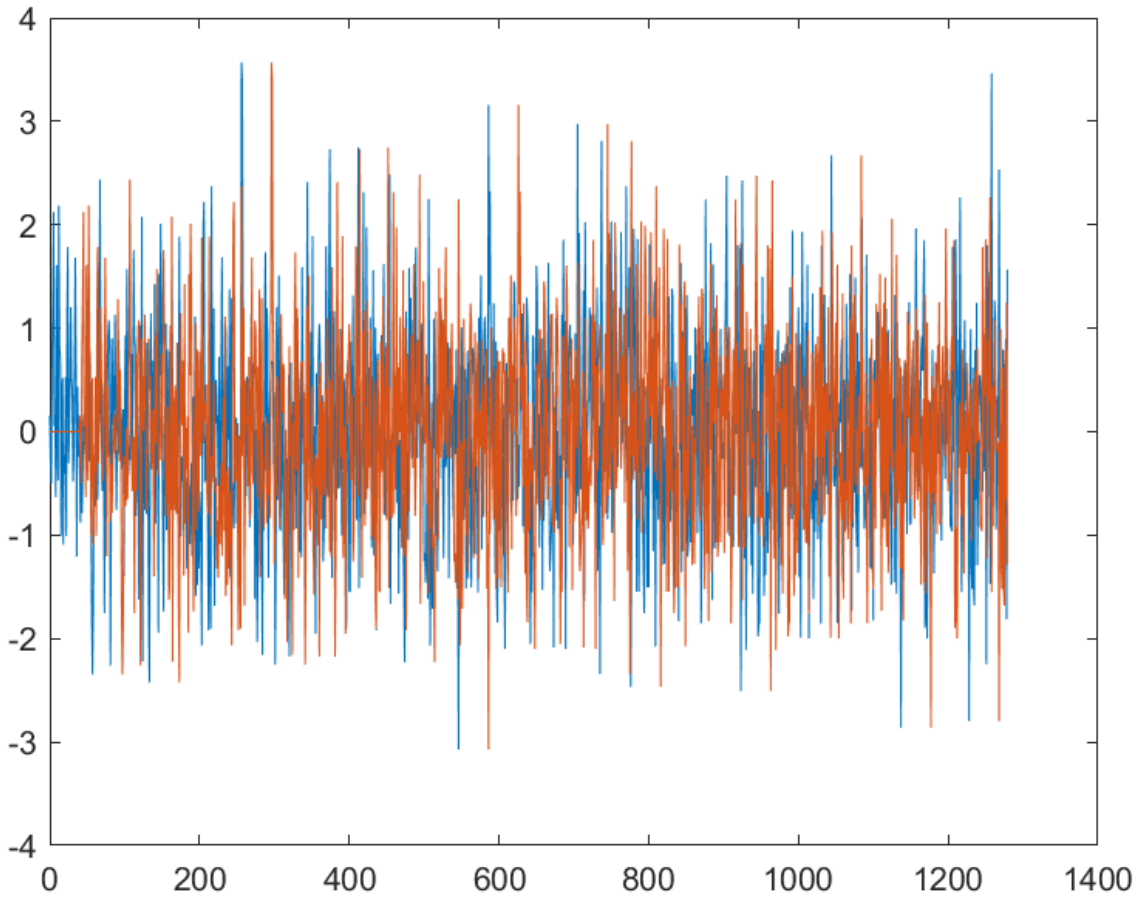

Figure: Simulation Results – Plot 2

FIR Filter

First, we initialize the state:

fir_init.m

```
function [state] = fir_init(h, Ns)
% [state] = fir_init(h, Ns);
%
% Creates a new FIR filter.
%
% Inputs:
% h Filter taps

% Ns Number of samples processed per block
% Outputs:
% state Initial state
%% 1. Save parameters
state.h = h;
state.Ns = Ns;
%% 2. Create state variables
% Make buffer big enough to hold Ns+Nh coefficients. Make it an
%integer power
% of 2 so we can do simple circular indexing.
% Temporary storage for circular buffer
% Set initial tail pointer and temp pointer (see pseudocode)
state.M = 2^(ceil(log2(state.Ns+1)));
```

```
state.Mmask = state.M-1;
% Temporary storage for circular buffer
state.buff = zeros(state.M, 1);
state.n_t = Ns;
state.n_p = state.n_t - 1;
```

Then, we develop the filter function:

fir.m

```
function [state_out, y] = fir(state_in, x)
% [state_out, y] = fir(state_in, x);
%
% Executes the FIR block.
%
% Inputs:
% state_in Input state
% x Samples to process
% Outputs:

% state_out Output state
% y Processed samples
% Get state
s = state_in;
% Move samples into tail of buffer
% Filter samples and move into output
% Return updated state
for ii=0:s.Ns - 1
 % Store a sample
 s.buff(s.n_t+1) = x(ii+1);
 % Increment head index (circular)
 s.n_t = bitand(s.n_t+1, s.Mmask);
  s.ptr = bitand(s.n_t+s.Mmask, s.Mmask);

 sum = 0;
 for g = 0:size(s.h)-1
    sum = sum + s.buff(s.ptr + 1) * s.h(g+1);
      s.ptr = bitand(s.ptr+s.Mmask, s.Mmask);
 end
 y(ii + 1) = sum;
end
state_out = s;
```

Lastly, we test the function using the following code:

Test_fir1.m

```
% test_fir1.m
%
% Script to test the FIR filter.
% Global parameters
Nb = 100; % Number of buffers
Ns = 100; % Samples in each buffer
% Generate filter coefficients
p.beta = 0.5;
p.fs = 0.1;
```

```
p.root = 0; % 0=rc 1=root rc
M = 64;


Nd = 10; % Delay of block

[h f H Hi] = win_method('rc_filt', p, 0.2, 1, M, 0);
% Generate some random samples.
x = randn(Ns*Nb, 1);
% Type of simulation
stype = 1; % Do simple convolution
%stype = 1; % DSP-like filter
if stype==0
 y = conv(x, h);
elseif stype==1
 % Simulate realistic DSP filter
 % ADD YOUR CODE HERE !!!

    state_fir = fir_init(h, Ns);

    % Generate some random samples.
    x = randn(Ns*Nb, 1);
    % Reshape into buffers
    xb = reshape(x, Ns, Nb);
    % Output samples
    yb = zeros(Ns, Nb);

    for bi=1:Nb
     [state_fir, yb(:,bi)] = fir(state_fir, xb(:,bi));
    end
    % Convert individual buffers back into a contiguous signal.
    y = reshape(yb, Ns*Nb, 1);

else
 error('Invalid simulation type.');
end
% Compute approximate transfer function using PSD
Npsd = 200; % Blocksize (# of freq) for PSD
[Y1, f1] = periodogram(y, [], Npsd,1);
[X1, f1] = periodogram(x, [], Npsd,1);


figure(1);
plot(f1, abs(sqrt(Y1./X1)), f, abs(H));
hold on;
grid on;
%plot(f, abs(H), 'b');
xlim([0 0.2]);
%ylim([0, 20]);
```

With stype = 0, a simple convolution takes place, which looks as follows:

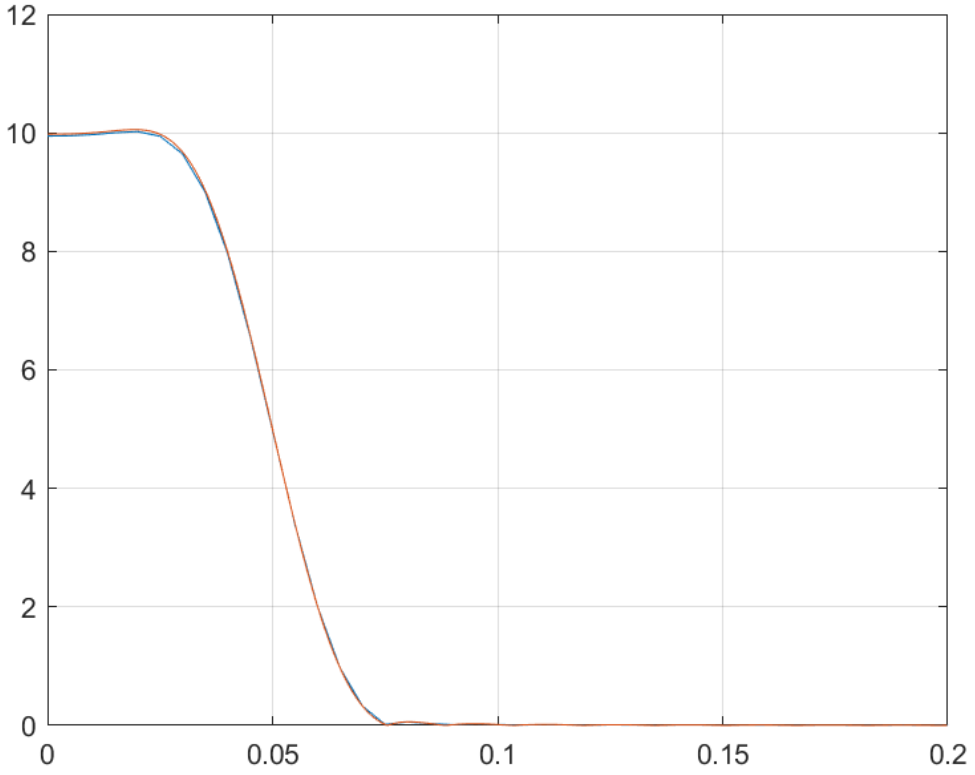

Figure: Convolution simulation

With stype = 1, the DSP like filter works as follows:

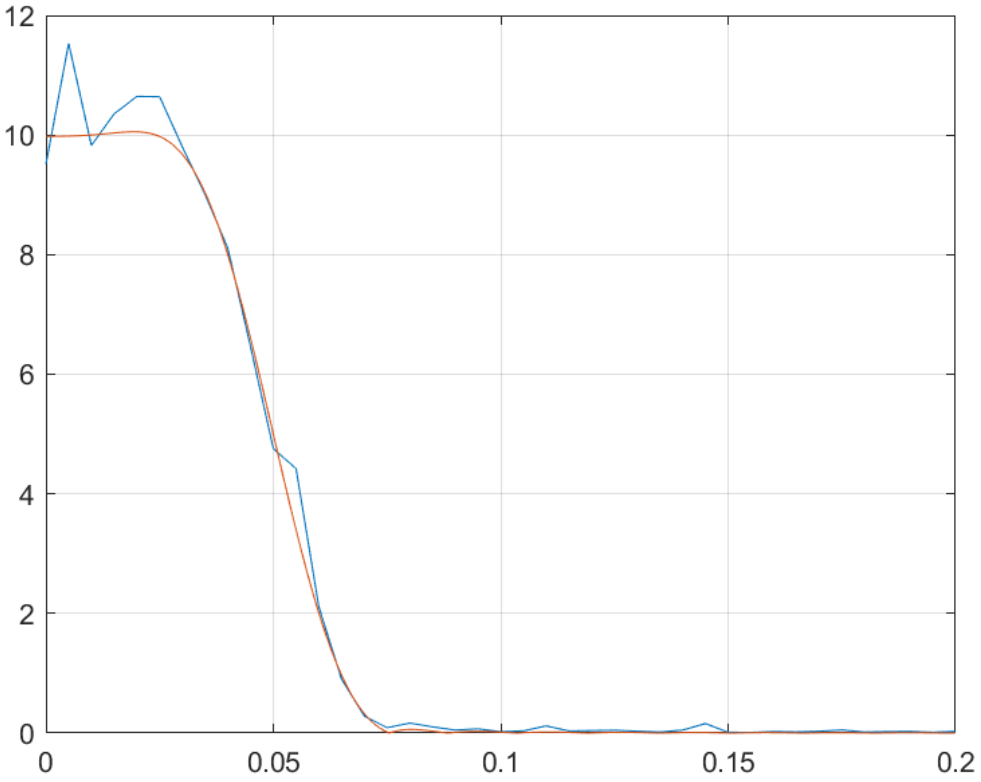

Figure: DSP like filter simulation

Lab Write-Up

1. A short explanation of the difference between a usual MATLAB simulation and real-time DSP code. Also, briefly explain why buffer-oriented processing is needed and what affects the latency and throughput of an algorithm.

For MATLAB simulation, we had all input samples available at the beginning of simulation, so the algorithm's functioning is independent of change in time – time is not a critical factor for proper functioning of the algorithm. However, in the case of real-time DSP code, a continuous stream of data is received in real-time, which also needs to be processed in real-time. Hence, in this case, the process is dependent on time and time of input arrival factors into the functioning of the algorithm.

Buffer oriented processing:

All samples are not readily available in a realistic situation. As a result, incoming chunks of signals are divided into a buffer of size 'N', which we can use to generate N output samples. Therefore, we use buffer-oriented programming in order to implement this design.

Latency is directly proportional to buffer size. Therefore, a small buffer size corresponds to low latency as not many buffers are filled up. But small buffer size means we are processing many small chunks – which means time would be wasted in storing and restoring state, which implies lower overall throughput.

The optimal value for N is decided through application constraints: the number of buffers is inversely proportional to throughput. Therefore, the higher the number of buffers, the higher the time loss in storing and restoring state and vice versa.

2. A diagram showing conceptually how your Delay uses a circular buffer to implement the Delay.

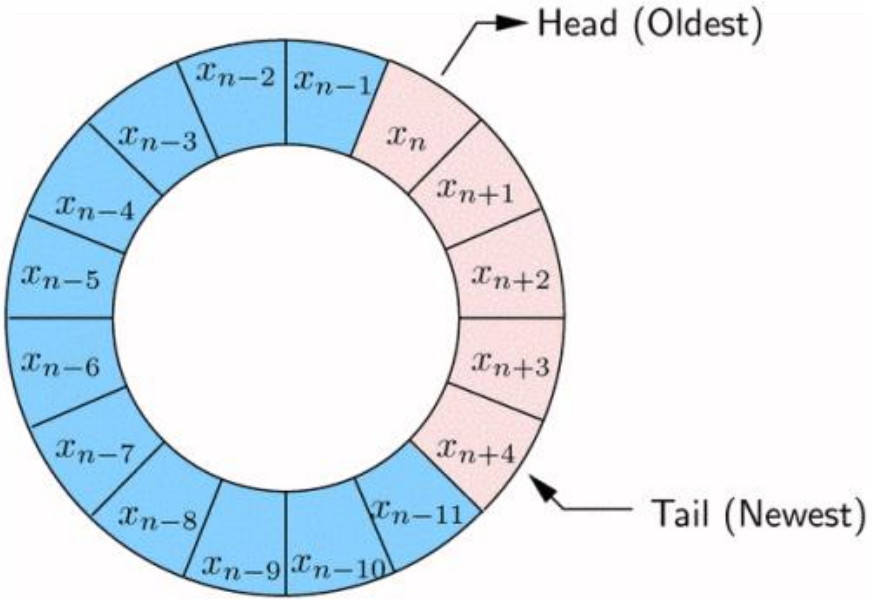

Figure: Circular buffer concept

The elements are arranged in a circle. New data is placed at the tail. The head points to the oldest item not yet processed. After a process is complete the head is incremented. When a new item is added to memory, the tail is incremented.

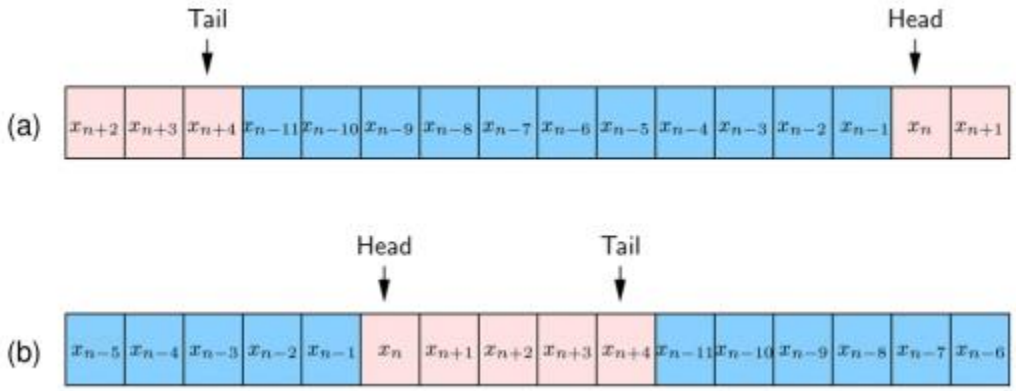

Figure: Circular buffer implementation – flattened out

The circular concept is accomplished through the flat structure by the use of a wrapping. When head or tail is at the end of the allotted memory, it is shifted to the beginning of the queue.

3. A printout of the MATLAB code that implements your Delay with comments. For FIR lab, provide the following items in the write up:

Delay Block

The source code for each part of the delay block is provided below:

We initialize the state using the following code:

delay_init.m

```
function [state] = delay_init(Nmax, N)
% [state] = delay_init(Nmax, N);
%
% Initializes a delay block.
%
% Inputs:
% Nmax Maximum delay supported by this block.
% N Initial delay
% Outputs:
% state State of block
% Notes:
% For this block to operate correctly,
% you should not pass in more than Nmax
% samples at a time.
%% 1. Save parameters
state.Nmax = Nmax;
% Store initial desired delay.
state.N = N;
%% 2. Create state variables
% Make the size of the buffer at least twice of the maximum delay.
% Allows us to copy in and then read out in just two steps.
```

```
state.M = 2^(ceil(log2(Nmax))+1);
% Get mask allowing us to wrap index easily

state.Mmask = state.M-1;
% Temporary storage for circular buffer
state.buff = zeros(state.M, 1);
% Set initial head and tail of buffer
state.n_h = 0;
state.n_t = N;
```

Now, we develop the delay function:

delay.m

```
function [state_out, y] = delay(state_in, x)
% [state_out, y] = delay(state_in, x);
%
% Delays a signal by the specified number of samples.
%
% Inputs:
% state_in Input state
% x Input buffer of samples

% Outputs:
% state_out Output state
% y Output buffer of samples
% Get input state
s = state_in;
% Copy in samples at tail
for ii=0:length(x)-1
 % Store a sample
 s.buff(s.n_t+1) = x(ii+1);
 % Increment head index (circular)
 s.n_t = bitand(s.n_t+1, s.Mmask);
end
% Get samples out from head
y = zeros(size(x));
for ii=0:length(y)-1
 % Get a sample
 y(ii+1) = s.buff(s.n_h+1);
 % Increment tail index
 s.n_h = bitand(s.n_h+1, s.Mmask);
end
% Output the updated state
state_out = s;
```

Testing the delay function:

Test_delay1.m

```
% test_delay1.m
%
% Script to test the delay block. Set up to model the way
% samples would be processed in a DSP program.
% Global parameters
```

```
Nb = 10; % Number of buffers
Ns = 128; % Samples in each buffer
Nmax = 200; % Maximum delay
Nd = 10; % Delay of block
% Initialize the delay block
state_delay1 = delay_init(Nmax, Nd);

% Generate some random samples.
x = randn(Ns*Nb, 1);
% Reshape into buffers
xb = reshape(x, Ns, Nb);
% Output samples
yb = zeros(Ns, Nb);
% Process each buffer
for bi=1:Nb
 [state_delay1, yb(:,bi)] = delay(state_delay1, xb(:,bi));
end
% Convert individual buffers back into a contiguous signal.
y = reshape(yb, Ns*Nb, 1);
% Check if it worked right
n = [0:(length(x)-1)];
figure(1);
plot(n, x, n, y);
figure(2);
plot(n+Nd, x, n, y, 'x');
% Do a check and give a warning if it is not right. Skip first buffer
%in check
% to avoid initial conditions.
n_chk = 1+[Ns:(Nb-1)*Ns-1];
if any(x(n_chk - Nd) ~= y(n_chk))
 warning('A mismatch was encountered.');
end
```

The results of simulation have been provided below:

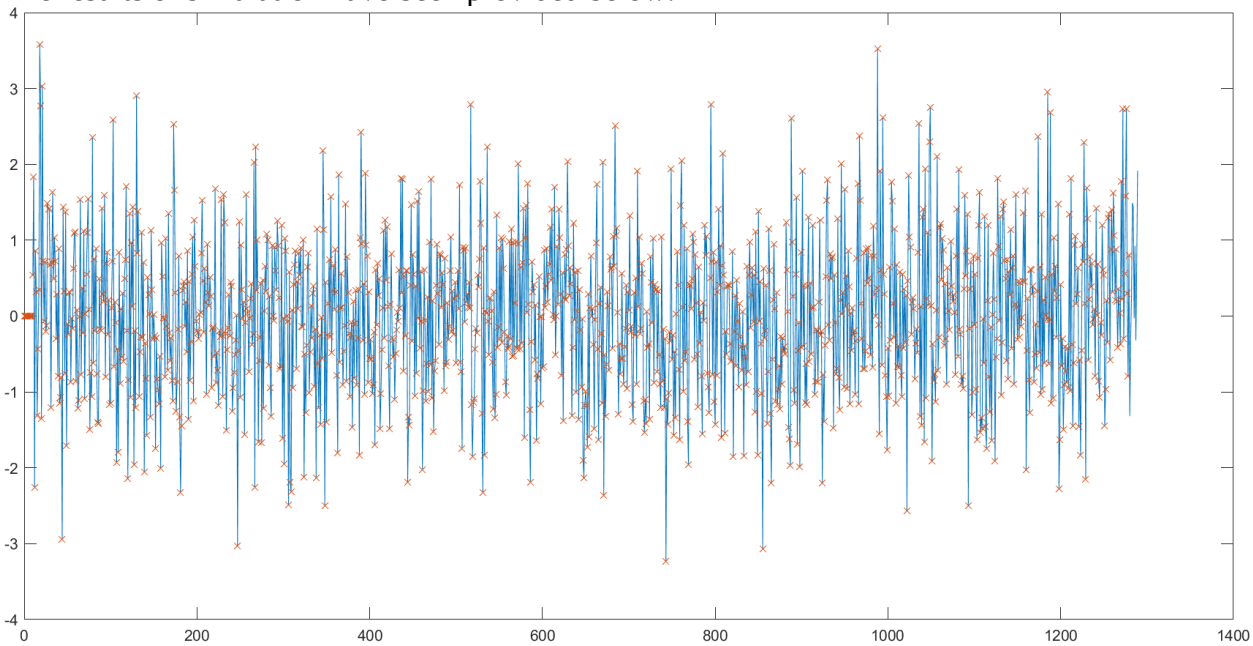

Figure: Simulation results – Plot 1

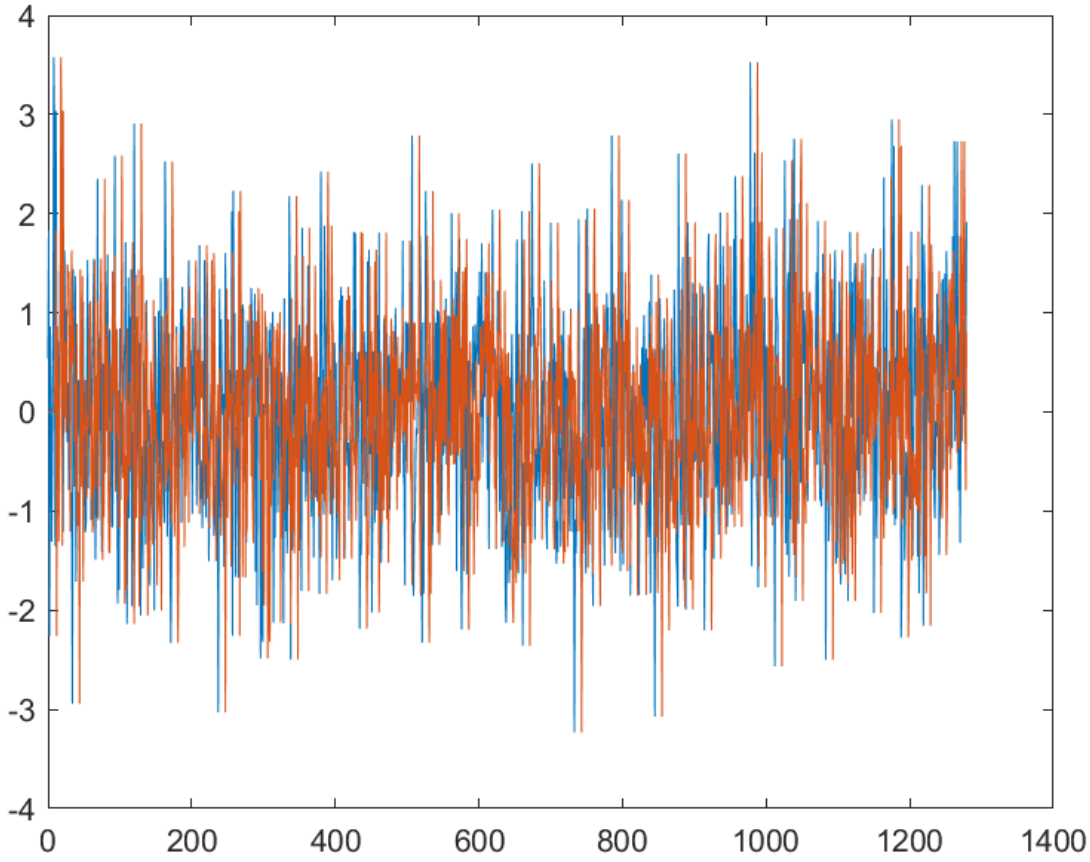

Figure: Simulation results – Plot 2

test_delay2.m

```
%
% Script to test the delay block. Set up to model the way
% samples would be processed in a DSP program.
% Global parameters
Nb = 10; % Number of buffers
Ns = 128; % Samples in each buffer
Nmax = 200; % Maximum delay
Nd = 20; % Delay of block
% Initialize the delay block
state_delay1 = delay_init(Nmax, Nd);
state_delay2 = delay_init(Nmax, Nd);
% Generate some random samples.
x = randn(Ns*Nb, 1);
% Reshape into buffers
xb = reshape(x, Ns, Nb);
% Output samples
yb = zeros(Ns, Nb);
yb2 = zeros(Ns, Nb);
% Process each buffer
for bi=1:Nb
 [state_delay1, yb(:,bi)] = delay(state_delay1, xb(:,bi));
 [state_delay2, yb2(:,bi)] = delay(state_delay2, yb(:,bi));
end
% Convert individual buffers back into a contiguous signal.
```

```
y = reshape(yb2, Ns*Nb, 1);
% Check if it worked right
n = [0:(length(x)-1)];
figure(1);
plot(n, x, n, y);
figure(2);
plot(n+Nd+Nd, x, n, y, 'x');
% Do a check and give a warning if it is not right. Skip first buffer
%in check
% to avoid initial conditions.
n_chk = 1+[Ns:(Nb-1)*Ns-1];
if any(x(n_chk - Nd) ~= y(n_chk))
 warning('A mismatch was encountered.');
end
```

The simulation results have been provided below:

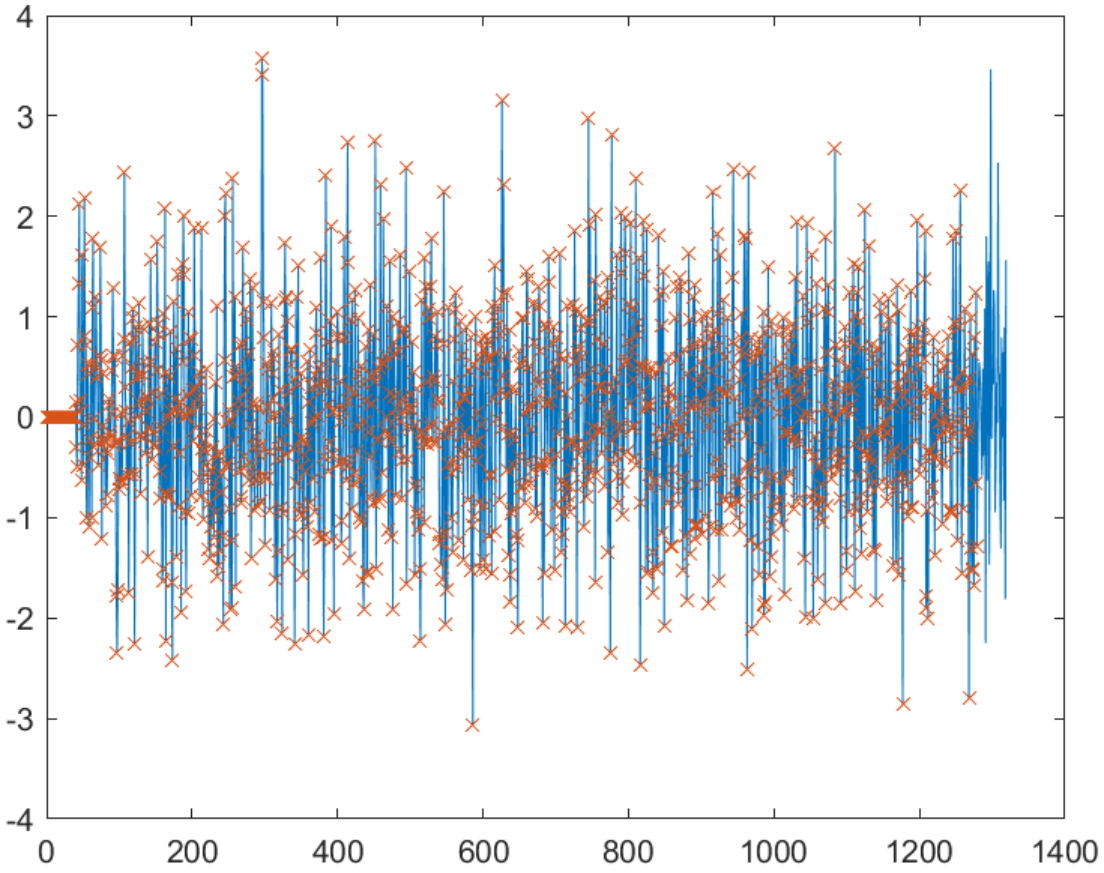

Figure: Simulation Results – Plot 1

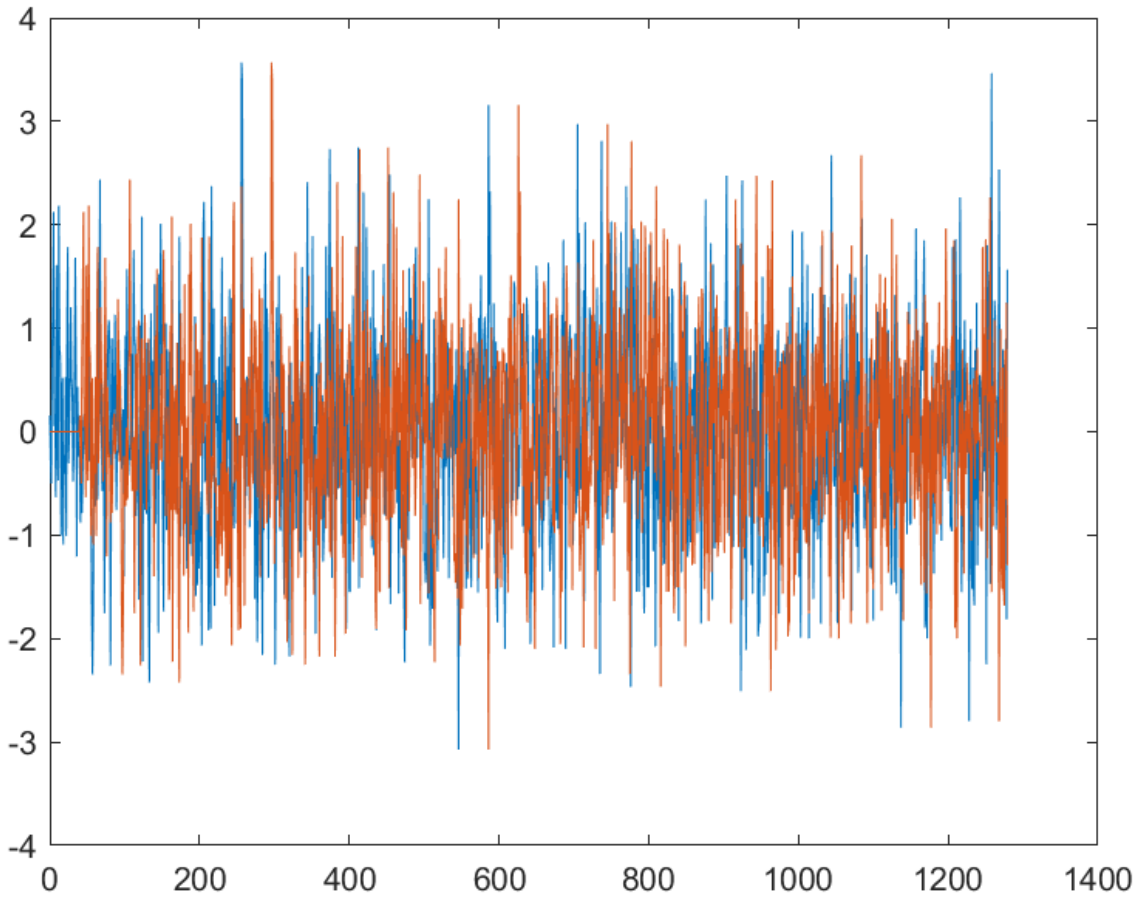

Figure: Simulation Results – Plot 2

1. A diagram showing conceptually how your FIR filter uses a circular buffer to implement the FIR equation.

The FIR equation is presented as follows:

$$y_n = \sum_{m=0}^{N-1} h_m x_{n-m}$$
$$= h_0 x_n + h_1 x_{n-1} + h_2 x_{n-2} + \dots$$

The block diagram for the filter implementation is provided below:

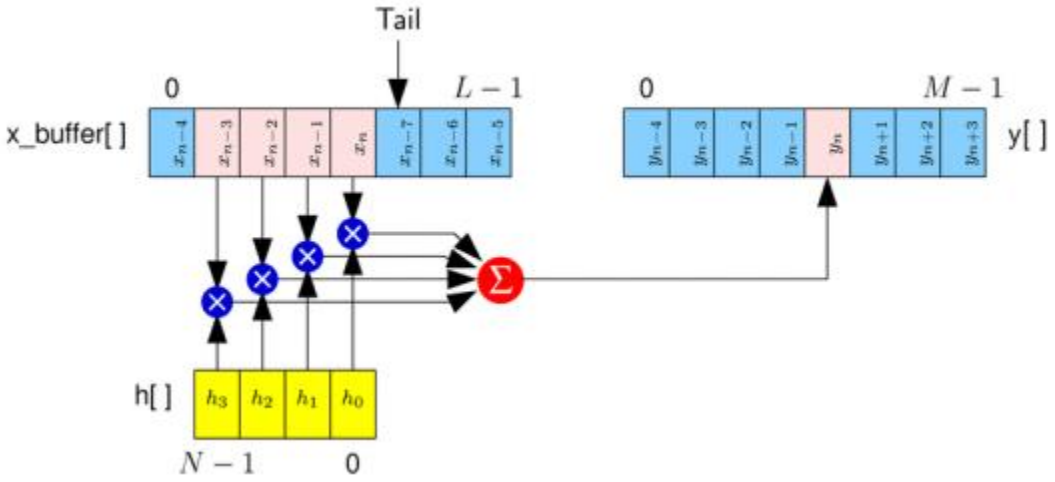

Figure: Block Diagram of FIR using buffer

The FIR implementation uses a buffer to develop an FIR filter.

The buffer receives N samples, which are moved to the circular buffer. Then, the samples are filtered to generate N output samples. Lastly, the N samples are copied into the output buffer.

In the above implementation, the circular buffer is flattened out. Conceptually, we perceive the circular buffer to look as follows:

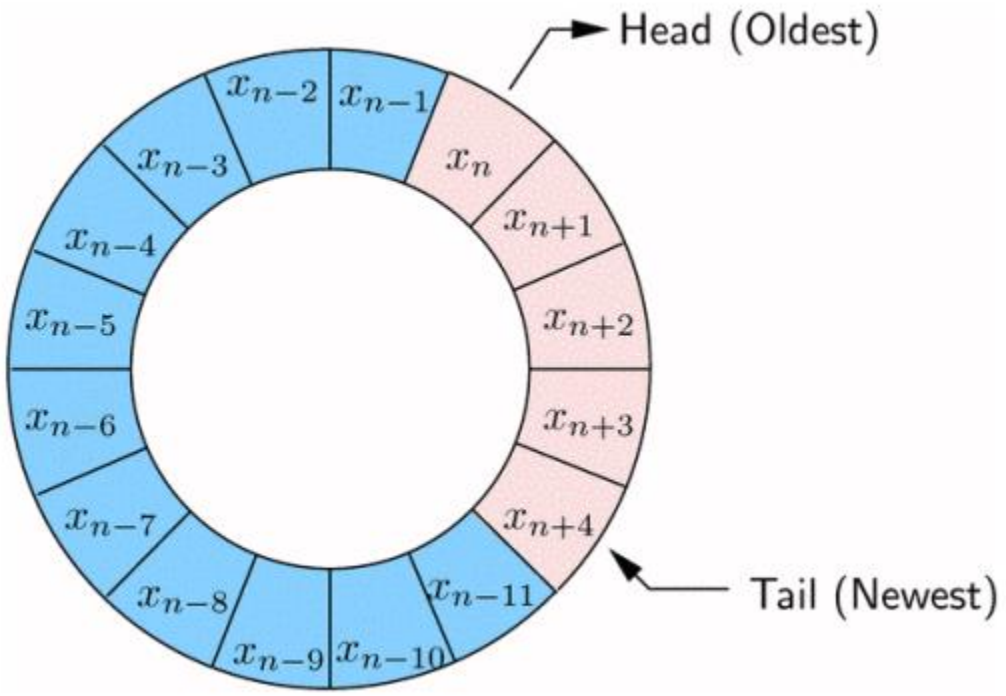

Figure: Circular buffer concept

When flattened out, the buffer looks as follows:

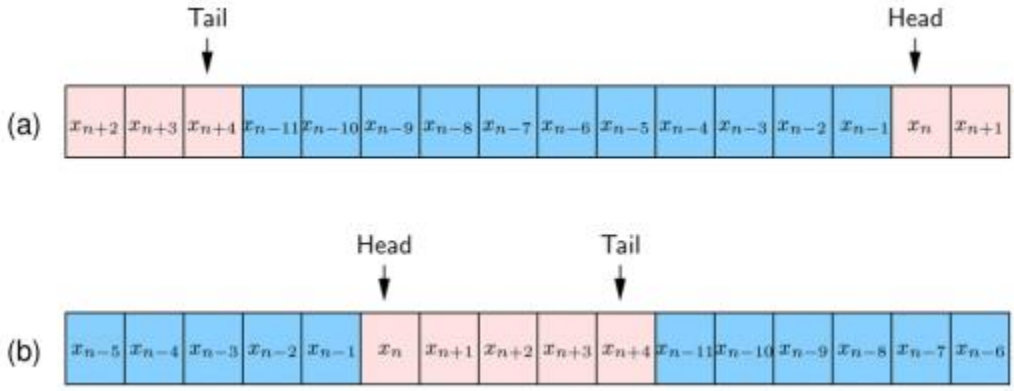

Figure: Flattened out circular buffer

2. A printout of the MATLAB code that implements your FIR filter with comments.

FIR Filter

First, we initialize the state:

fir_init.m

```
function [state] = fir_init(h, Ns)
% [state] = fir_init(h, Ns);
%
% Creates a new FIR filter.
%
% Inputs:
% h Filter taps

% Ns Number of samples processed per block
% Outputs:
% state Initial state
%% 1. Save parameters
state.h = h;
state.Ns = Ns;
%% 2. Create state variables
% Make buffer big enough to hold Ns+Nh coefficients. Make it an
%integer power
% of 2 so we can do simple circular indexing.
% Temporary storage for circular buffer
% Set initial tail pointer and temp pointer (see pseudocode)
state.M = 2^(ceil(log2(state.Ns+1)));
state.Mmask = state.M-1;
% Temporary storage for circular buffer
state.buff = zeros(state.M, 1);
state.n_t = Ns;
state.n_p = state.n_t - 1;
```

Then, we develop the filter function:

fir.m

```
function [state_out, y] = fir(state_in, x)
% [state_out, y] = fir(state_in, x);
%
% Executes the FIR block.
%
% Inputs:
% state_in Input state
% x Samples to process
% Outputs:

% state_out Output state
% y Processed samples
% Get state
s = state_in;
% Move samples into tail of buffer
% Filter samples and move into output
% Return updated state
for ii=0:s.Ns - 1
 % Store a sample
 s.buff(s.n_t+1) = x(ii+1);
 % Increment head index (circular)
 s.n_t = bitand(s.n_t+1, s.Mmask);
  s.ptr = bitand(s.n_t+s.Mmask, s.Mmask);

 sum = 0;
 for g = 0:size(s.h)-1
    sum = sum + s.buff(s.ptr + 1) * s.h(g+1);
      s.ptr = bitand(s.ptr+s.Mmask, s.Mmask);
 end
 y(ii + 1) = sum;
end
state_out = s;
```

Lastly, we test the function using the following code:

Test_fir1.m

```
% test_fir1.m
%
% Script to test the FIR filter.
% Global parameters
Nb = 100; % Number of buffers
Ns = 100; % Samples in each buffer
% Generate filter coefficients
p.beta = 0.5;
p.fs = 0.1;
p.root = 0; % 0=rc 1=root rc
M = 64;


Nd = 10; % Delay of block

[h f H Hi] = win_method('rc_filt', p, 0.2, 1, M, 0);
% Generate some random samples.
x = randn(Ns*Nb, 1);
```

```
% Type of simulation
stype = 1; % Do simple convolution
%stype = 1; % DSP-like filter
if stype==0
 y = conv(x, h);
elseif stype==1
 % Simulate realistic DSP filter
 % ADD YOUR CODE HERE !!!

    state_fir = fir_init(h, Ns);

    % Generate some random samples.
    x = randn(Ns*Nb, 1);
    % Reshape into buffers
    xb = reshape(x, Ns, Nb);
    % Output samples
    yb = zeros(Ns, Nb);

    for bi=1:Nb
     [state_fir, yb(:,bi)] = fir(state_fir, xb(:,bi));
    end
    % Convert individual buffers back into a contiguous signal.
    y = reshape(yb, Ns*Nb, 1);

else
 error('Invalid simulation type.');
end
% Compute approximate transfer function using PSD
Npsd = 200; % Blocksize (# of freq) for PSD
[Y1, f1] = periodogram(y, [], Npsd,1);
[X1, f1] = periodogram(x, [], Npsd,1);


figure(1);
plot(f1, abs(sqrt(Y1./X1)), f, abs(H));
hold on;
grid on;
%plot(f, abs(H), 'b');
xlim([0 0.2]);
%ylim([0, 20]);
```

3. Plots showing the ideal response of your filter compared to the simulated response of the filter. Explain any discrepancies.

The ideal response of the filter looks as follows:

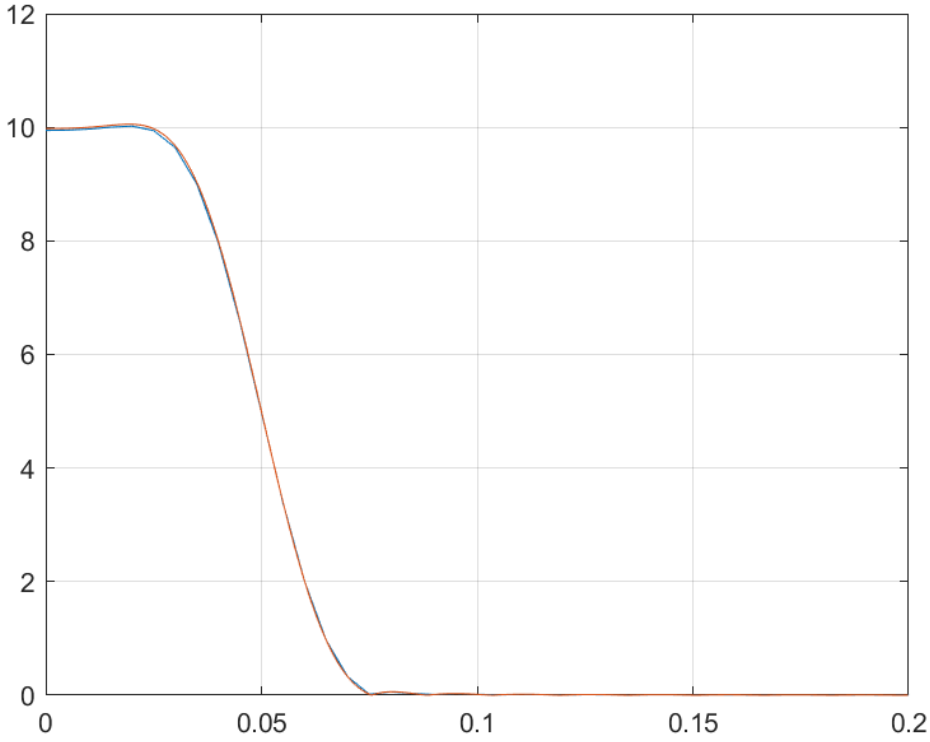

Figure: Convolution simulation

The simulated response looks as follows:

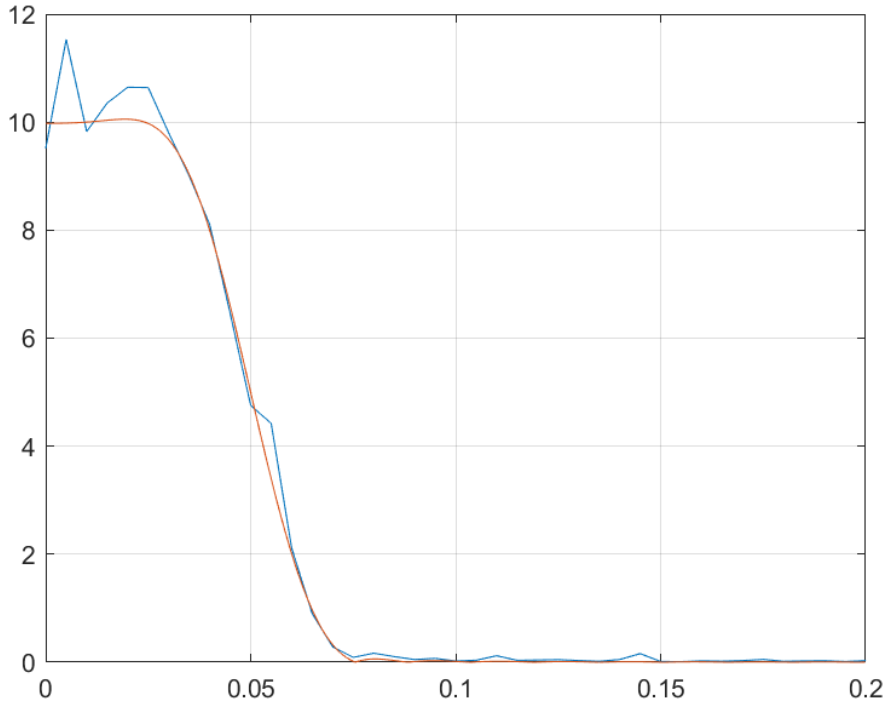

Figure: DSP like filter simulation

The ideal response is an exact match to the original data, so it has a smooth distribution that is superimposed onto the original curve. The simulated response, however, is not smoothly distributed over the original curve. Rather, it has fluctuations throughout the lifecycle of the signal, which shows that the simulated response is an approximation.

4. Any problems you ran into in the lab and how you fixed them

It was very difficult to figure out how the FIR worked. Understanding what parts of the filter were similar to the buffer, and which parts were different was also a challenge. How the state variable is required to be manipulated so that the FIR used the delay structure to develop a filter method took some time, but we were able to figure it out in the end.

Conclusion

In this lab, we worked on the Real-Time Block Simulation of Delay and FIR in MATLAB. We started with an understanding of the usage of buffers in different implementations in communications systems, and learned about the circular queue, which is used as a buffer in the implementations we worked on. We were provided some of the code for Delay and FIR implementations, and were required to complete it. On the first task, we learned how to use the buffer to implement a delay, then observed how the simulation results matched with our own understanding of the delay. For FIR, we learned about the Window method of FIR Filter Design. Then we implemented the filter using a Digital Filter Implementation method shown to us using a buffer. We were then able to run the code and compare our simulated response with the original response, which gave us an understanding of how an FIR filter works.

References

Lab Manual: Delay and FIR – Real-Time Simulation in MATLAB